Lightweight Silent Data Corruption Detection Based on Runtime Data Analysis for HPC Applications

Eduardo Berrocal* , Leonardo Bautista-Gomez† , Sheng Di† , Zhiling Lan* , and Franck Cappello†‡

*Illinois Institute of Technology, Chicago, IL, USA
†Argonne National Laboratory, Argonne, IL, USA
‡University of Illinois at Urbana-Champaign, Champaign, IL, USA
eberroca@iit.edu, leobago@anl.gov, sdi1@anl.gov, lan@iit.edu, cappello@anl.gov

***Abstract*—Next-generation supercomputers are expected to have more components and, at the same time, consume several times less energy per operation. Hence, supercomputer designers are pushing the limits of miniaturization and energy-saving strategies. Consequently, the number of soft errors is expected to increase dramatically in the coming years. While mechanisms are in place to correct or at least detect soft errors, a percentage of those errors pass unnoticed by the hardware. Such silent errors are extremely damaging because they can make applications silently produce wrong results. In this work we propose a technique that leverages certain properties of HPC applications in order to detect silent errors at the application level. Our technique detects corruption based solely on the behavior of the application datasets and is mostly algorithm-agnostic. We propose multiple corruption detectors, and we couple them to work together in a fashion transparent to the user. We evaluate our strategy on well-known HPC applications and kernels such as HACC and Nek5000. Our results show that some detectors can detect up to 95% of corruptions and other lightweight detectors can cover for the majority of corruptions while incurring less than 5% overhead.**

***Index Terms*—Fault Tolerance, Resilience, High-Performance Computing, Data Mining, Silent Data Corruption, Soft Errors, One-Step-Ahead Prediction, Time Series**

I. Introduction

High-performance computing (HPC) is changing the way scientists make discoveries. Science applications require ever-larger machines to solve problems with higher accuracy. While future systems promise to provide the power needed to tackle those science problems, they are also raising new challenges. For example, transistor size and energy consumption of future systems must be significantly reduced, steps that might dramatically impact the soft error rate (SER) according to recent studies [1], [2].

Random memory access (RAM) devices have been intensively protected against soft errors through error-correcting codes (ECCs) because they have the largest share of the susceptible surface on high-end computers. Recent studies, however, indicate that ECCs alone cannot correct an important number of DRAM errors [3]. In addition, not all parts of the system are ECC-protected: in particular, logic units and registers inside the processing units are usually not ECC-protected because of the space, time, and energy cost that ECC requires in order to work at low level. Historically, the SER of central processing units was minimized through a technique called *radiation hardening* [4], which consists of increasing the capacitance of circuit nodes in order to increase the critical charge needed to change the logic level. Unfortunately, this technique involves increasing either the size or the energy consumption of the components, which is prohibitively expensive at extreme scale. Thus, a non-negligible ratio of soft errors could pass undetected by the hardware, corrupting the numerical data of HPC applications. This is called silent data corruption (SDC).

In this work, we leverage the fact that the datasets produced by HPC applications (i.e., the applications' state at a particular point in time) have characteristics that reflect the properties of the underlining physical phenomena that those applications attempt to model. These characteristics can be used effectively to design a general SDC detection scheme with relatively low overhead. In particular, we propose to leverage the spatial and temporal behavior of HPC datasets to predict an interval of *normal* values for the evolution of the datasets, such that any corruption will *push* the corrupted data point outside the expected interval of *normal* values, and it will, therefore, become an *outlier*.

Building a lightweight and efficient SDC detector for HPC applications is a challenging endeavor. On the one hand, it is unclear what are the most effective techniques to monitor and predict HPC datasets' evolution. On the other hand, an HPC application's data (as well as its characteristics) usually change over time, enforcing any detector to dynamically adapt. Moreover, advanced prediction techniques require relatively large amounts of historic data and/or long training periods, which are not feasible in practice.

The contributions of this work are summarized as follows.

- We design a battery of SDC detectors, relying on several prediction methods, with different accuracy and cost levels, that leverage the properties of HPC datasets.
- We theoretically analyze different prediction cases in order to calculate optimal parameters for our detectors.
- We study the propagation of corruption on HPC appli-

cations, including the transfer to other processes.
- We perform a comprehensive evaluation using all our detectors with a number of popular HPC applications, and we show that our detectors can guarantee over 90% of SDC coverage on real application runs.
- We discuss the performance and memory overheads incurred by the proposed detectors and the trade-off between detection cost and accuracy.

The rest of the paper is organized as follows. In Section II we present related work. In Section III we present our proposed detectors. In Section IV we introduce our analytical model. In Section V we present our evaluation and results. In Section VI we summarize our key findings.

II. Related Work

The problem of data corruption for extreme-scale computers has been the target of numerous studies. They can be classified in three groups depending on their level of generality, that is, how easily a technique can be applied to a wide spectrum of HPC applications. They also have different costs in time, space, and energy. An ideal SDC detection technique should be as general as possible, while incurring a minimum cost over the application.

A. Hardware-Level Detection

The most general method is to try to solve the problem of data corruption at the hardware level. This method is extremely general because applications do not require any adaptation to benefit from such detectors. Considerable literature exists on soft errors rates [5], [6], [2], [7] and detection techniques at the hardware level [8], [9]. Implement these techniques efficiently under the strict constraints of extreme-scale computing (e.g., low power consumption) is difficult, however. Moreover, market interest in driving technologies in this direction is uncertain.

B. Process Replication

Process replication has been used for many years to guarantee correctness in critical systems, and its application to HPC systems has been studied. Fiala et al., for example, proposed using double-redundant computation to detect SDC by comparing the messages transmitted between the replicated processes [10]. The authors also suggested using triple redundancy to enable data correction through a voting scheme. This approach is general in that applications need little adaptation to benefit from double or triple redundancy. Their customized MPI implementation (RedMPI) assumes that corruption in application data manifests itself by producing different MPI messages between replicas. In [11], the authors take advantage of multithreading and multicore processors to replicate threads of execution, so faults can be detected by comparing outputs from replicated threads. Unfortunately, double- and triple-redundant computation always imposes large overheads, since the number of hardware resources will be doubled or tripled. In addition, the cost of energy consumption is heavily increased when using full replication, not only because of the extra computation, but also because of the extra communications. In contrast to process replication, our techniques do not incur any network overhead, and their memory footprints are always below 100%.

C. Algorithm-Based Fault Tolerance

A promising technique against data corruption is algorithm-based fault tolerance (ABFT) [12]. This technique uses extra checksums in linear algebra kernels in order to detect and correct corruptions [13]. However, ABFT is not general, since each algorithm needs to be adapted by hand, and only some linear algebra kernels have been adapted, which is only a subset of the vast spectrum of computational kernels. Furthermore, even applications that employ only ABFT-protected kernels could fail to detect SDCs if the corruption lies *outside* the ABFT-protected regions. In comparison, our proposed approach is general enough to protect any memory region of any HPC application.

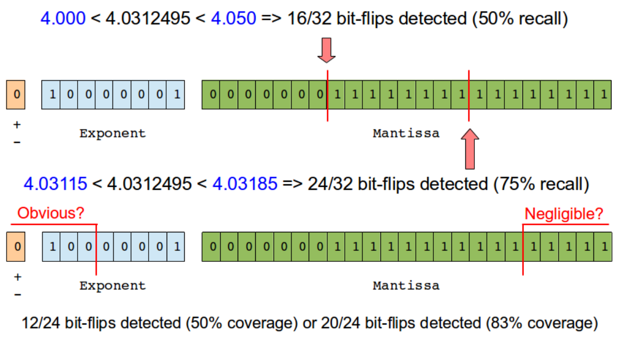
D. Approximate Computing

Another type of SDC detection is based on the idea of approximate computing. In this detection method, a computing kernel is paired with a cheaper and less accurate kernel that will produce *close enough* results. Such results can be compared with those generated by the main computational kernel [14]. This detection mechanism shows promising results, but again it is still not general enough, since each application needs to be manually complemented with the required approximate computing kernels. Furthermore, complex applications also need to adapt multiple kernels to offer good coverage.

III. Anomaly Detection

In this work we propose to use data mining to detect SDC during runtime in scientific applications. We believe that a strategy based on data analytics is an interesting path to explore for several reasons. First, such an approach is completely independent of the underlying algorithm and therefore dramatically more general than algorithm-based techniques. Second, one can develop lightweight data-monitoring techniques that impose a low overhead on the application compared with that from extremely expensive techniques such as double and triple redundancy. Third, data monitoring and outlier detection can be offered by the runtime in a fashion transparent to the user. *Anomaly detection* has been used in multiple domains such as medical analysis.

Our main idea is to monitor the application datasets during runtime in order to predict an interval of *coherent* values for the next time step and then raise alerts when some data points go outside this range (*outliers*). For instance, assume a dataset that is evolving during execution and is being monitored by our detector. The detector analyzes the dataset



4.000 < 4.0312495 < 4.050 => 16/32 bit-flips detected (50% recall)
0
1 0 0 0 0 0 0 1
0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
+
-
Exponent
Mantissa
4.03115 < 4.0312495 < 4.03185 => 24/32 bit-flips detected (75% recall)
Obvious?
Negligible?
12/24 bit-flips detected (50% coverage) or 20/24 bit-flips detected (83% coverage)

Figure 1. Anomaly detection based on prediction.

at each time step and predicts an expected range of values for the next time step. As shown in Figure 1, the detector predicts that normal values should be inside the segment $[4.000, 4.050]$. A data point with the value 4.0312495 is inside that range. If that data point were corrupted in one of the 16 most significant bits of the IEEE floating-point representation, the value would automatically move outside the expected range of normal values and would be detected as an outlier. Now, let us imagine that our detector could give a more accurate prediction, giving $[4.03115, 4.03185]$ as the interval of normal values. In this case, any corruption in the 24 most significant bits would push the point outside the new and narrower interval.

We evaluate the efficacy of our detectors using two well-known metrics: precision (denoted ρ) and recall (denoted τ), defined in Equations (1) and (2), respectively. Here *TP*, *FP*, and *FN* refer to *True Positives*, *False Positives*, and *False Negatives*, respectively. As shown above, the accuracy of the prediction has a direct impact on the detection recall.

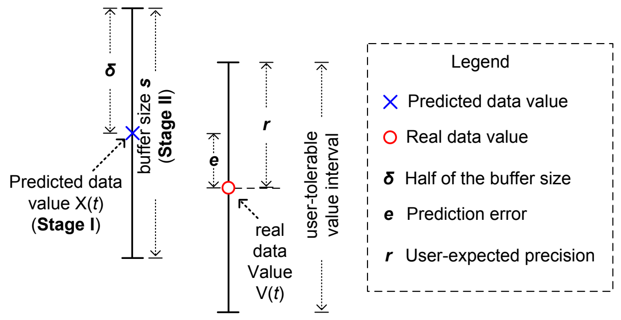
$$\rho = \frac{TP}{TP + FP} \tag{1}$$

$$\tau = \frac{TP}{TP + FN} \tag{2}$$

Another important point is that not all the bits in the IEEE floating-point representation need to be covered. For instance, a corruption in the most significant bits is likely to generate numerical instability, inducing the application to crash. Such soft errors might be silent to the hardware but not to the application. On the other hand, corruption in the least significant bits of the mantissa might produce deviations that are lower than the allowed error of the application and hence, are negligible. On the second detector discussed above, if we neglected the 4 most-significant bits (numerical instability) and the 4 least-significant bits (negligible error), we could say that the detector has a coverage of 20 bits out of 24 (83% coverage). In the following subsections we introduce multiple detectors with different accuracy and cost levels.

A. One-Step Ahead Linear Predictors

In this section we introduce our anomaly detectors based on point wise time evolution prediction. Here we try to closely follow the time evolution of each data point in the domain in order to predict the value at the next time step. Our point wise SDC detection approach has two steps: a step that involves the prediction of the next expected value in the time series for each data point; and another step which determines a buffer (i.e., normal value interval) surrounding the predicted next-step value. Soft errors can be detected by observing whether a particular value falls outside of this computed buffer. Comparing with many prediction methods, our approach needs to perform only one-step ahead prediction using recent data values to achieve high accuracy. This is because adjacent time steps show high data correlation. The buffer size will play an important role for the expected precision and recall. The optimal buffer size depends on the relative location of the predicted value and the user-expected accuracy. In general, all HPC applications have different expected accuracy for its computation results. For example, users may expect their accuracy to always be within 10^{-8}. Hence, for each computed data point, there will be a user-expected (or tolerable accuracy) value interval. In Section IV we present an analysis about the optimal buffer size for different predicted values and user-expected accuracy.



δ
buffer size s (Stage II)
Predicted data value X(t) (Stage I)
e
r
real data Value V(t)
user-tolerable value interval
Legend
× Predicted data value
○ Real data value
δ Half of the buffer size
e Prediction error
r User-expected precision

Figure 2. Illustration of one-step prediction model (at time step t).

The key notation used to formulate the detection model is presented in Figure 2. The buffer used to detect silent errors is denoted by $[X(t) - \delta, X(t) + \delta]$, where $X(t)$ is the predicted value at time step t. The buffer size is denoted by $s = 2\delta$. In general, the magnitude of the prediction error will depend on the prediction method used, as shown in Figure 2. In this figure, the real data value is presented as a red circle, and the prediction error (denoted by e) is equal to the difference between the predicted value $X(t)$ and the real data value (denoted by $V(t)$) computed at the current time step. The user-expected (or tolerated) value interval is denoted by $[V(t) - r, V(t) + r]$, as shown in figure 2. Now, we introduce a number of different predictors that are the core of our point wise anomaly detectors. Every predictor involves a trade-off between overhead and prediction error.

1) Linear Curve Fitting: Our first predictor, called linear curve fitting (*LCF*), uses the two most recent previous time steps to fit a linear curve, which is then projected to the next time step in order to predict the next value in the time series. Equation (3) shows how this prediction is calculated. Δ_{t-1} is the slope of the curve (velocity) at time $t-1$.

$$\begin{aligned} X(t) &= \Delta_{t-1} + V(t-1) \\ &= (V(t-1) - V(t-2)) + V(t-1) \\ &= 2V(t-1) - V(t-2) \end{aligned} \tag{3}$$

2) Acceleration-Based Predictor: The acceleration-based predictor (*ABP*) uses the two and three most recent previous time steps to extract the velocity (Δ_{t-1}) and acceleration (Δ^2_{t-1}) of the data, respectively, and then combines them to compute the prediction for the next value in the time series.

By definition we have

$$\begin{aligned} \Delta^2_{t-1} &= \Delta_{t-1} - \Delta_{t-2} \\ \Delta_{t-1} &= V(t-1) - V(t-2) \\ \Delta_{t-2} &= V(t-2) - V(t-3). \end{aligned}$$

Putting these together results in

$$\begin{aligned} X(t) &= \Delta^2_{t-1} + \Delta_{t-1} + V(t-1) \\ &= 3V(t-1) - 3V(t-2) + V(t-3). \end{aligned} \tag{4}$$

3) Auto Regressive Model: The auto regressive model (*AR*) assumes that every value in the time series depends linearly on its previous values. Equation (5) describes AR, where c is a constant, φ_i are the coefficients of the model, p the number of coefficients, and ε_t the noise at time t. Normally, we can assume $\varepsilon_t \sim \mathcal{N}(0, \sigma^2)$. The coefficients φ_i are computed by using the first 10 time steps of the simulation (we assume that no errors occur during this period) by least squares with the Yule-Walker equations.

$$X(t) = c + \sum_{i=1}^{p} \varphi_i V(t-i) + \varepsilon_t \tag{5}$$

Clearly, the value chosen for p is critical. A large value of p may give a better prediction, but it will incur a high memory footprint. We note that as opposed to LCF and ABP, which keep only past values V, AR needs to keep both the coefficients φ_i and the past values V in memory. In our detector, we set $p = 4$.

4) Auto Regressive Moving Average Model: In the auto regressive moving average (*ARMA*) model, we add the polynomial from the moving average (MA) model to the AR model. In ARMA, we need to specify how many coefficients we have from both the AR model (p), and the MA model (q). For our detector, we use $p = 4$ and $q = 4$. The errors ε_{t-i} are computed by using the past prediction errors.

$$\varepsilon_{t-i} = V(t-i) - X(t-i)$$

The model is described as follows.

$$X(t) = c + \sum_{i=1}^{p} \varphi_i V(t-i) + \varepsilon_t + \sum_{i=1}^{q} \theta_i \varepsilon_{t-i} \tag{6}$$

B. Cluster-Based Anomaly Detector

In this section, we expand the idea of time-based prediction to a spatial and spatiotemporal predictors in order to use multiple detectors simultaneously. First we introduce an optimization technique to limit the overhead induced by the point wise prediction methods. In contrast to the point wise detectors (see Section III-A), here we envision a scheme based on predicting the evolution of a *cluster* of nearby points. That is, all the data will be classified into a number of clusters based on vicinity. The features of each cluster, such as the probability density function (PDF), will be extracted, monitored, and used for prediction. Such a design will be inexpensive because we do not need to monitor each data point independently but only treat the cluster features instead. We use γ to denote the data in the cluster; each element in γ is called a *feature point*. Any feature point outside of the expected distribution will be considered an outlier.

1) Dataset Distribution: The first cluster-based detector we propose is to estimate the normal value interval for the target dataset γ that the next-step data will fall inside with highest probability. Based on the PDF at time step ($t-1$) and the evolution of the PDFs at previous time steps (such as $t-2$), this detector gives a prediction of the possible PDF for the detection time step (t). Any observed data point located outside the boundaries of the predicted PDF will be treated as an outlier. We call this detector a $\gamma - detector$.

2) Spatial Anomaly Detector: The second detector we propose analyzes the space variations of the dataset γ. It computes at each time step $t-1$ a distribution for β (beta), where β is computed as shown in Equation (7) for a 2D domain.

$$\begin{aligned} &\beta_{(i,j)}(t) = \gamma_{(i,j)}(t) - \gamma_{(g,h)}(t) \\ &where\ \ g, h \in \{i-1; i; i+1\} \end{aligned} \tag{7}$$

This computation can take into account the neighbors in one or more dimensions, as well as many neighbor points (e.g., 5-point stencil) depending on the preferences of the user. Then, the detector produces and stores the PDF of the β for the last time step $t-1$ and predicts a PDF for the next detection time step t. As in the previous case, any feature point indicating that $\beta_{(i,j)}(t)$ is outside the expected range of normal values is treated as an outlier. We call this detector a $\beta - detector$.

3) Temporal Anomaly Detector: The third type of detector that we have developed is based on the temporal evolution of a dataset. For each feature point, we compute the difference ϵ (epsilon) as shown in Equation (8).

$$\epsilon_{(i,j)}(t) = \gamma_{(i,j)}(t) - \gamma_{(i,j)}(t-1) \tag{8}$$

Then, as with other detectors, we compute the distribution of ϵ at time step $t-1$, and check the observed values at next time step t. Any $\epsilon_{(i,j)}(t)$ that violates the predicted PDF is treated as an outlier. Computing $\epsilon(t)$, however, requires saving $\gamma(t-1)$ in memory, thus involving an extra memory overhead. To avoid this overhead, we sacrifice the accuracy to a certain extent by leveraging the index of PDF. Specifically, we split the domain value range [min_value, max_value] evenly into 256 bins. Instead of keeping $x_{(i,j)}(t-1)$ for each point in the domain, we keep only a 1-byte word that indexes the value closest to $\gamma_{(i,j)}(t-1)$ from among the 256 bins. In this way, we reduce the memory footprint of such detector from 4 or 8 bytes (for single or double precision, respectively) per data point to only 1 byte. We call this detector an $\epsilon - detector$.

4) Spatiotemporal Anomaly Detector: The fourth detector we propose in this work is a spatiotemporal detector that computes the time evolution (denoted by ζ (zeta)) of the β, as shown in Equation (9).

$$\zeta_{(i,j)}(t) = \beta_{(i,j)}(t) - \beta_{(i,j)}(t-1) \tag{9}$$

Computing the time gradient of the space gradient gives us an idea of when a dataset increases or decreases its level of *turbulence*. Similar to the temporal anomaly detector, one must keep β values of the previous time step in order to compute the time difference. Thus, we employ the same indexing technique (loosing a little accuracy) to reduce the overhead from 4 or 8 bytes per data point to only 1 byte. We call this detector a $\zeta - detector$.

IV. Analysis of the Detection Cases and Optimization of Buffer Size

In this section, we theoretically analyze different prediction cases – in terms of prediction error and user expected precision – in order to calculate an optimal buffer for our predictors.

A. Analysis of Silent Error Detection Cases

A total of six cases are used for the relative locations and/or values of the detection buffer and user-tolerable value interval, as shown in Figure 3. In every case, the value space is splitted into five different parts, each of which corresponds to a particular detection result. In case 1 and case 6, the estimated buffer stays completely outside the user-expected interval; in case 2 and case 5, the buffer and the user-expected interval overlap to a certain extent; in case 3, the buffer completely falls inside the user-expected value interval, so no FN detections occur; in case 4, the buffer contains the user-expected interval, so again there are no FP detections.

B. Optimizing Buffer Sizes for One-Step-Ahead Prediction

In this subsection, we optimize the buffer sizes based on different cases as shown in Figure 3. That is, our objective is to optimize the value of δ, given some conditions required

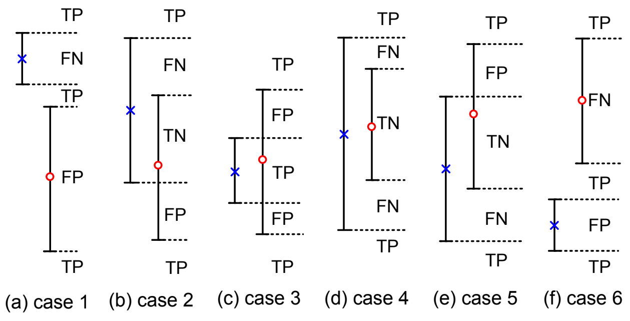


Figure 3. Location analysis of buffer vs. user-tolerable interval

by users. We use $\delta^*_{\{condition\}}$ to denote the optimal δ when being subject to a particular condition (e.g., ρ=100%).

For simplicity, we discuss the δ^* only when ρ=100% or τ=100% is expected/required by users.

Theorem 1: When ρ=100% or τ=100%, the following three propositions hold, provided that r and e are constants.

① $\delta^*_{\{\rho=100\%\}} = r + e$
② $\delta^*_{\{e \le r, \tau=100\%\}} = r - e$
③ $\delta^*_{\{e > r, \tau=100\%\}}$ does not exist.

Proof: There are two cases: $e \le r$ or $e > r$.

As for $e \le r$ (such as Figure 3(b)-(e)), the estimated buffer must overlap the user-expected value interval. Five different situations exist, as shown in Figure 4.

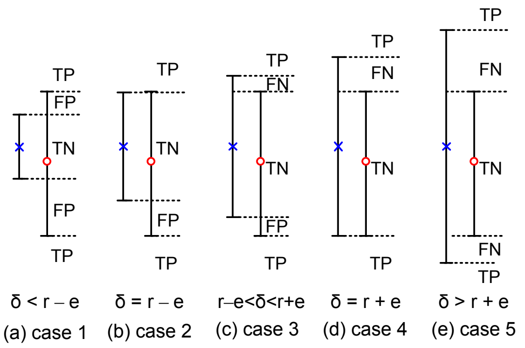


Figure 4. The Five situations subject to $e \le r$

The predicted value $X(t)$ is fixed because of the constants e and r, while the buffer sizes increase through Figure 4 (a)-(e). Obviously, $\tau = 100\%$ can occur only in case 1 and case 2 because there must be no FN detections for both of these two cases. That is, for meeting τ=100%, $\delta \le r - e$ must hold. By comparing Figure 4(a) and (b), we can see that the number of TP detections is the same while the FP detections in case 2 is never greater than that of case 1. As a result, case 2 leads to the largest precision under the condition $e \le r$. In other words, Equation (10) (proposition ②) holds.

$$\delta^*_{\{e \le r, \tau=100\%\}} = r - e \tag{10}$$

Similarly, we can derive Equation (11) from Figure 4 (d) and (e), and Equation(12) using a similar analysis for $e > r$.

$$\delta^*_{\{e \le r, \rho=100\%\}} = r + e \tag{11}$$

$$\delta^*_{\{e > r, \rho=100\%\}} = r + e \tag{12}$$

Combining Equation (11) and (12), we get proposition ①.

To illustrate proposition ③, let us go back to Figure 3(a). We see that, in order to guarantee a τ=100%, we would have to let δ=0; otherwise, s(=2δ)$\neq$0, which means that there would be detection cases where FN > 0. However, δ=0 (or $s = 0$) conflicts with the checkpoint/restart model since it would report all values in all time steps as outliers. This would, in turn, produce a rollback in every time step, making the progression of the application's execution impossible. ■

In practice, the next-step real data value $V(t)$ is unknown, thus the next-step prediction error, e, is unknown, too. However, we can estimate the confidence prediction error interval (i.e., the maximum prediction error, denoted by e_{max}) as well as the expected prediction error (denoted by $\bar{e}$).

Theorem 2 further specifies the interval for the optimal values of δ^*, with only prediction error conditions (i.e., $e \le r$), and its expected value.

Theorem 2: The following three formulas hold.

$$r - e_{max} \le \delta^*_{\{e \le r\}} \le r + e_{max} \tag{13}$$

$$E(\delta^*_{\{\rho=100\%\}}) = r + \bar{e} \tag{14}$$

$$E(\delta^*_{\{e \le r, \tau=100\%\}}) = r - \bar{e} \tag{15}$$

Proof:

(1) *Proving Equation (13)*

On the one hand, we can prove that for any particular prediction error e ($\le r$), $r - e \le \delta^* \le r + e$ must hold. Such a proposition can be proved by using Figure 3, which shows all possible cases where the prediction error e is less than or equal to the user-expected interval r. We can see that precision and recall in case 2 can never be smaller than in case 1. The reason is that FN $= 0$ holds in both cases but FP is greater in case 1 than in case 2. Hence, $\delta^*_{e \le r} \ge r - e$ holds. Similarly, we can prove that $\delta^*_{e \le r} \le r + e$ based on Figure 3 (d) and (e).

On the other hand, we have that $e \le e_{max}$ ($\forall e$), therefore we get inequality 16 with respect to any error e ($\le r$).

$$r - e_{max} \le r - e \le \delta^*_{\{e \le r\}} \le r + e \le r + e_{max} \tag{16}$$

Consequently, Equation (13) holds.

(2) *Proving Equations (14) and (15)*

According to Theorem 2, we have that $\delta^*_{\{\rho=100\%\}} = r + e$ and $\delta^*_{\{e \le r, \tau=100\%\}} = r - e$. In this proof, we use x to denote the random variable associated with the prediction error (where $x \le e_{max}$), and we use $f(x)$ to denote the probability density function. We can compute the expected δ^* (denoted by $E(\delta^*)$) as follows

$$\begin{aligned}
E(\delta^*_{\{\rho=100\%\}}) &= \int_0^{e_{\max}} f(x)(r + x)dx \\
&= r + \int_0^{e_{\max}} f(x)xdx \\
&= r + \bar{e} \\
E(\delta^*_{\{e \le r, \tau=100\%\}}) &= \int_0^{e_{\max}} f(x)(r - x)dx \\
&= r - \int_0^{e_{\max}} f(x)xdx \\
&= r - \bar{e}
\end{aligned}$$
■

C. *Optimizing Buffer Size for Cluster-Based Detector*

To optimize the buffer size for cluster-based detector, we must take into account the changing trend of the estimated bounds, that is, the increase/decrease of the normal value interval bounds generated based on the analysis of the past data/features. The reason is that the group features of the current step data during the execution may not always comply with the features at last step in practice. Suppose that the estimated intervals at last two steps ($t-2$ and $t-1$) are computed as [$lb(t-2)$,$ub(t-2)$] and [$lb(t-1)$,$ub(t-1)$], respectively, where *lb* and *ub* refer to lower bound and upper bound, respectively. Then, the estimated bounds at the time t will be set as follows

$$lb(t) = 2 \cdot lb(t-1) - lb(t-2) \tag{17}$$

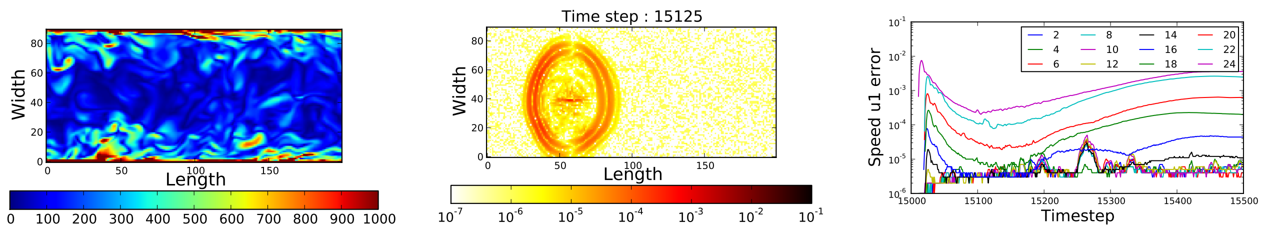
$$ub(t) = 2 \cdot ub(t-1) - ub(t-2) \tag{18}$$

This approach is the one we follow for our cluster-based detectors in order to avoid a large number of false alerts.

V. Evaluation

In this section, we present a set of experiments performed to test the efficacy of our SDC detector in production-level HPC applications. All of our detectors, as well as our bitflip injector, are implemented transparently inside the fault tolerance interface (FTI) [15], originally used by applications to perform efficient checkpoints of chosen datasets. The only requirement for applications to use the SDC detectors inside FTI is to add an extra library call in the main loop.

To cover a wide range of possible HPC datasets, we use a computational fluid dynamics (CFD) mini app [16], Nek5000 [17] (a CFD kernel), and HACC [18] (an N-body cosmology application). The CFD mini app simulates a turbulent flow in a 3D duct modeled as a large eddy simulation (LES) using a two-stage time-differencing scheme based on higher accuracy for compressible gas using Navier-Stokes equations. The 3D duct is divided in N sections along the length (x axis) of the duct, where N corresponds to the number of MPI ranks in the execution. LES codes are among the most challenging applications in this context because of their chaotic and hard-to-predict behavior. They also represents a large set of HPC applications, ranging from weather prediction to aerospace engineering. Nek5000 is a CFD solver based on the spectral element method. It



(a) Vorticity in turbulent fluid (b) Error propagation of 24th bit corruption (c) Maximum deviation after corruption

Figure 5. Error propagation in turbulent flow simulation

is also being used for a large number of applications in diverse fields such as reactor thermal-hydraulics and biofluids. HACC (for Hybrid/Hardware Accelerated Cosmology Code) is a cosmology code aimed at understanding the nature of dark matter and dark energy in the universe. It uses N-body methods and is optimized for a wide range of systems (including those using accelerators). HACC divides the computation of the gravitational force into two phases: a long/medium range spectral particle-mesh (PM) component, which is common to all architectures, and an architecture-tunable particle-based short/close-range solver.

A. Corruption Propagation

We start by analyzing how corruption propagates in classic HPC applications, such as the CFD code mentioned above. CFD applications produce vorticity plots to show the turbulence of the fluid. For instance, Figure 5(a) shows the vorticity of the fluid on a 2D cut of the 3D duct. However, the vorticity is computed from the velocity fields in the three axes and is never stored in a variable. Therefore, an error deviation observed in the vorticity plot is likely to be the consequence of a corruption happening in one of the velocity fields. In this run, we injected a bit-flip (grid point $40X40$) in the 24th bit position, the first bit of the exponent. This corruption is barely visible to the naked eye if plotted on a figure. Yet this corruption will generate large perturbations that will propagate across the domain, reaching other MPI ranks and corrupting the large majority of the domain.

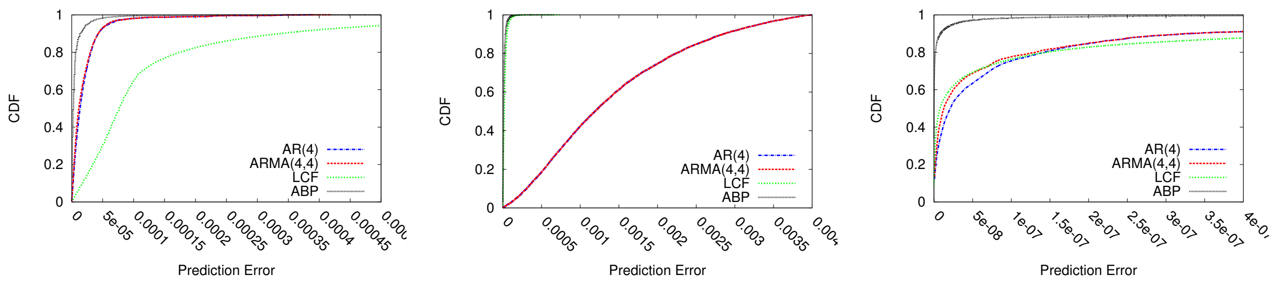
To study the propagation of corruption after an SDC, we performed the following experiment. First, we launched a turbulent flow simulation starting from the initial conditions and let it run for 15,000 time steps to let the gas reach a high level of turbulence. Then, we restarted the execution from time step 15,000 using FTI, and we recorded the datasets of the execution at each time step for a corruption-free execution. We confirmed that several corruption-free execution produce identical results. Then, we repeated the same experiment but this time injecting one bit-flip at bit position $2p$ for p in [1,12]. For each experiment we injected the bit-flip in the first twenty time steps and let it run for 500 iterations.

After all the corruption experiments were done, we computed for each experiment and for each time step the difference between the corrupted dataset and the corruption-free dataset. Figure 5(b) plots this deviation a hundred time steps after corrupting the 24th bit of the grid point $40X40$. We use a logarithmic color scale to show the magnitude of the deviation in the different regions of the domain. As we can observe, in only a hundred iterations the corruption has already propagated across the entire domain, and it shows a particularly high deviation in a region with a wave shape that has as origin the corrupted grid point. Although the origin of the corruption was in the grid point $40X40$, the fluid has moved in those hundred time steps, and the corruption wave's epicenter is about 20 grid points to the right, which happens to be located in an MPI rank other than the one where the corruption was originally injected.

We plotted similar figures for every time step of each corruption experiment [19], but here, for brevity, we show only one. We note that the high deviation wave bounces in the walls of the duct and continues propagating in other directions. To get an idea of how deviations behave for different corruption levels, we plotted the maximum deviation at each time step for all the corruption experiments. As we can see in Figure 5(c), during the first ten time steps or so, no corrupted data occurs. When the bit-flip is injected, we observe a sudden jump with a magnitude exponentially proportional to the bit-flip position, which is consistent with the floating-point representation. In addition, we notice that immediately after the corruption jump, the deviation starts to decrease. This decrease is due to a *smoothening* effect that takes place when the noncorrupted data interacts with the corrupted data. However, the same influence can go in the other direction. For instance, when the corruption wave bounces in the wall of the duct, it interacts with the other part of the wave that is just arriving to the wall, generating a corruption amplification effect, which is what we see happening after time step 15,150. Finally, the deviation stabilizes around time step 15,400 and remains stable until the end of the execution.

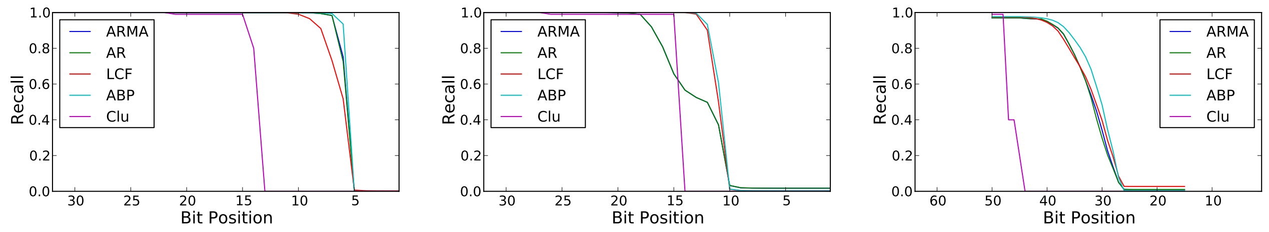
B. Prediction Errors for One-Step Ahead Linear Predictors

As we have seen in Section IV, the optimal value of the buffer size ($s^* = 2\delta^*$) depends on the prediction error e. To get an idea of the magnitude of this error, we run a set of experiments using different predictors in traces generated from the three mentioned HPC applications: one of the position's coordinates (x) of the particles in HACC, the vertical flow for Nek5000, and velocity for Turbulence-CFD.



(a) HACC (particles' position) (b) Turbulence-CFD (velocity) (c) Nek5000 (vortex)

Figure 6. CDF of prediction errors for different predictors and HPC datasets.



(a) HACC (particles' position) (b) Turbulence-CFD (velocity) (c) Nek5000 (vortex)

Figure 7. Recall for bit-flips injected on applications' traces.

These traces represent millions of prediction errors, which allow us to build a cumulative distribution function (CDF) of the size of e, as shown in Figure 6.

The most interesting result from these experiments is that a relatively simple predictor such as *ABP* is able to achieve smaller prediction errors than more complex linear models uch as the well-known *AR* and *ARMA* models. In absolute terms, for the HACC application, up to 90% of predictions have an error lower or equal to 0.000014 under ABP, where only 8% to 56% of predictions can reach such low errors under other predictors. For Nek5000, the prediction errors can be reduced to 8×10^{-9} for 90% of predictions. Moreover, the *AR* and *ARMA* models require not only more memory sizes per data point but also a parameter learning phase.

These experiments, however, do not help us in setting a good value for δ. Note that the actual prediction error e at a time step t is not known at runtime, making impossible to compute the optimal value for δ. Instead, we use the expectation of our desired optimal delta as shown in Equation (14). Thus, in order to compute $\bar{e}$, we assume errors close in time also experience a high degree of autocorrelation (as does data itself). In this way, $\bar{e}$ gets estimated by using the prediction error in last time step, as shown as follows.

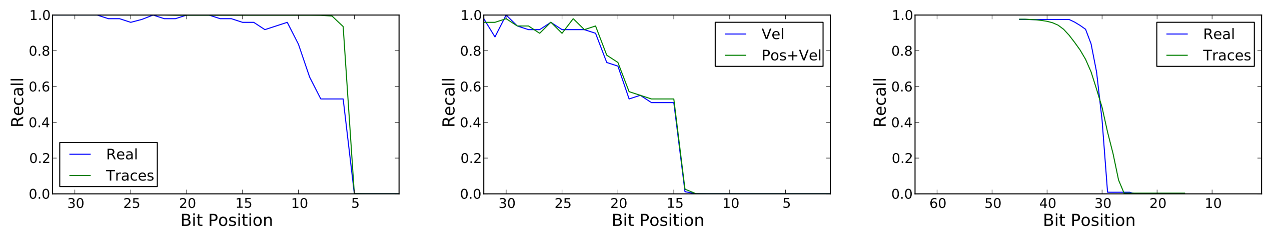
$$\bar{e}_t = |V(t-1) - X(t-1)| \tag{19}$$

C. Detection Results on Traces

In the first set of experiments, presented in Figure 7, we run our detectors using traces extracted from our three selected HPC datasets. We set the buffer size with the help of our analytical model (see Section IV) to maximize precision in order to avoid an excessive number of FP, which could render our detectors prohibitively expensive, assuming a FP always trigger extra actions (e.g., application-aware data consistency checks). For instance, in the checkpoint/restart model, detectors with poor precision could produce highly frequent rollbacks, making the execution progression impossible.

We notice in these results that the cluster-based anomaly detector *Clu* (see Section III-B), with less than 25% of memory overhead per data point, can guarantee a large coverage (over 50% of recall) against SDC. Moreover, we note that for these HPC applications, the protected variables represent less than 25% of the used memory, which translates into less than 5% of memory footprint for the cluster-based detectors. Therefore, these detectors can cover for the majority of corruptions for a negligible overhead.

We also observe that the time-based point wise detectors achieve the highest recall compared with the cluster-based detector. This is not surprising given the amount of memory overhead per data point of these predictors: 300% for LCF, 400% for ABP, 800% for AR, and 1600% for ARMA; as well as the more expensive computation involved. In particular, *ABP* achieves the best recall among all the other detectors, which is consistent with the prediction error results presented in Section V-B. For *ABP* we see an overall coverage (all bit positions included) above 60% for the Turbulence-CFD and above 80% for HACC or above 95% if we consider errors in the first 5 bits of the mantissa to be negligible. In the case of Nek5000, we can detect more than 75% of the corruptions for bit-flips on bit positions ≥ 33. Although these point wise detectors have a large memory footprint, they could be useful for low-memory footprint applications or for applications that are willing to pay the memory cost in order to have a high confidence that their results are corruption-free.



(a) HACC (particles' position) (b) HACC (velocity) (c) Nek5000 (vortex)

Figure 8. Comparing recall for bit-flips injected during real executions.

D. Detection Results at Runtime

In the second set of experiments, we test our detectors in real application runs. The purpose of these experiments is to study how runtime detection results compare to detection results on traces. We choose the ABP detector because it has the highest recall. We use HACC and Nek5000 as candidate applications. We run HACC with 512 MPI ranks and around 16 million particles, protecting the position and velocity variables. Nek5000 is run with 64 MPI ranks and a grid of 573,440 data points per rank. The variables protected are 7: position(x,y,z), velocity(vx,vy,vz), and pressure.

In Figure 8 we inject bitflips at random particles on particular bit positions on different datasets. We consider any detection occurring five time steps after the corruption (including the injection time step) as a true positive. In the figure, *vel* refers to injection and detection on the particles' velocity dataset, and *pos+vel* refers to injection on velocity while *detecting on position*. In the latter case (Figure 8(b)), we wanted to explore the idea of leveraging datasets' correlation for detection (i.e., making a corruption in one dataset visible by the other). In the case of HACC, *velocity* is used to move a particle to a new *position*.

Three conclusions can be extracted from these results: First, if we consider the first five bits of the mantissa to be negligible, our method can cover over 90% of all possible corruptions for the position dataset in HACC (Figure 8(a)). Similarly, if we consider the first 15 bits of the mantissa to be negligible, our method can cover 75% of corruptions for the vortex dataset in Nek5000 (Figure 8(c)).

Second, the performance of our detector depends heavily on the underlying dataset (Figure 8(b)). We have observed that position changes are smoother than velocity changes, thus making the next values for velocity more difficult to predict. In fact, we have observed that prediction errors for velocity are an order of magnitude higher than those for position. The good news is that our idea of leveraging datasets correlations works. Apart from the savings in memory overhead, these results indicate that we can achieve a similar recall monitoring only position, than monitoring both.

Third, we see that our predictors have different results depending on whether we work with traces or real application runs. The reason for such disparities is that traces do not represent the totality of the application's data state, and are used only to construct distributions to help us understand different predictors and parameters. In any case, the results are indeed similar enough to make us confident in our experiments using traces.

We also performed experiments for other detectors (e.g., cluster based detectors) showing the significance of the trace based results, but, for brevity, we do not plot those figures.

E. Performance Overheads

In this section we analyze the cost of our different detectors in relation to their performance. By using the *ABP* detector and assuming we protect all datasets in the HACC application, the overheads imposed on the application are 84% extra memory consumption, 13.75% extra computation time, and 0% extra network communication. These overheads are calculated for a relatively small HACC run (512 MPI ranks). However, the fact that our approach does not have any network overhead makes it automatically scalable to larger runs. We expect that the extra computation time will decrease as more ranks and larger datasets are introduced.

A 84% memory overhead might be acceptable for applications that are not memory bound like Nek5000. However, it might be too expensive for applications like HACC that are memory bound (in particular, at extreme scale). In such cases, we recommend using cluster-based detectors that, with a memory and computation overhead under 5% (in the case of HACC application), can still guarantee over 50% of coverage, or over 60% if we assume the last 5 bits of the mantissa to be negligible. For the CFD code, the ABP detector improves the recall only by 11% in comparison with cluster-based detectors, while imposing 16 times more memory consumption per data point. Hence, cluster-based detectors are more cost-effective.

VI. Conclusion and Future Work

In this work we have presented a novel approach to tackle the problem of SDC in HPC applications. We propose to detect SDC by taking advantage of the characteristics of HPC' applications' datasets. We have thereby designed a large battery of SDC detectors with different costs and accuracy levels, and developed an analytical model to help us tune those detectors to achieve almost perfect precision. We implemented and evaluated our detectors with production-level scientific applications using both traces and real experiments on supercomputers. Some new insights and key findings are summarized below.

- *Error Propagation Study*: We show that corruption on some bit positions might be negligible, as injection in those bits does not deviate over a certain treshold.
- *Prediction Accuracy of One-step Ahead Predictors*: For HACC, up to 90% of predictions have an error lower or equal to 0.000014 under ABP, in comparison to only 8% to 56% for other predictors. For Nek5000, the prediction errors can be reduced to 8×10^{-9} for 90% of predictions.
- *Detection with Injected Errors*: ABP detectors lead to the highest recall: up to 95% of SDC can be covered with almost perfect precision in some cases.
- *Overhead*: Cluster-based detectors are the most cost-effective detectors, covering the majority of corruptions (over 50% of recall) for a negligible cost (less than 5% of overhead in some cases).

Our work extends our previous results in that domain [20], [21] and opens new research opportunities in the area of SDC avoidance for HPC. We plan to explore other lightweight detection methods from data analytics, the combination of these methods and the introduction of data semantics aspects in the analysis.

VII. Acknowledgments

This work is supported by the Argonne National Laboratory, the U.S. Department of Energy, the Office of Science and the Advanced Scientific Computing Research Program, under Contract DE-AC02-06CH11357, and also in part by the ANR RESCUE, and INRIA-Illinois Joint Laboratory for Petascale Computing.

References

[1] D. Li, J. S. Vetter, and W. Yu, "Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 57:1–57:11.

[2] S. Borkar, "Designing reliable systems from unreliable components: The challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, pp. 10–16, Nov. 2005.

[3] A. A. Hwang, I. A. Stefanovici, and B. Schroeder, "Cosmic rays don't strike twice: Understanding the nature of dram errors and the implications for system design," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'XVII)*. NY, USA: ACM, 2012, pp. 111–122.

[4] S. Krishnamohan and N. R. Mahapatra, "Analysis and design of soft-error hardened latches," in *Proceedings of the 15th ACM Great Lakes Symposium on VLSI (GLSVLSI'05)*. New York, NY, USA: ACM, 2005, pp. 328–331.

[5] E. Normand, "Single event upset at ground level," *IEEE Transactions on Nuclear Science*, vol. 43, no. 6, pp. 2742–2750, 1996.

[6] T. Semiconductor, "Soft errors in electronic memory - a white paper," 2004.

[7] Cataldo, "Mosys, iroc target ic error protection," 2002. [Online]. Available: http://www.eetimes.com/story/OEG20020206S0026

[8] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, "The soft error problem: An architectural perspective," in *11th International Symposium on High-Performance Computer Architecture (HPCA'05)*, 2005, pp. 243–247.

[9] T. J. Dell, "A white paper on the benefits of chipkill-correct ecc for pc server main memory," *IBM Microelectronics Division*, pp. 1–23, 1997.

[10] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell, "Detection and correction of silent data corruption for large-scale high-performance computing," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 78:1–78:12.

[11] S. Mukherjee, M. Kontz, and S. Reinhardt, "Detailed design and evaluation of redundant multi-threading alternatives," in *Proceedings of 29th Annual International Symposium on Computer Architecture*, 2002, pp. 99–110.

[12] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Transactions on Computers*, vol. 100, no. 6, pp. 518–528, 1984.

[13] Z. Chen, "Online-abft: an online algorithm based fault tolerance scheme for soft error detection in iterative methods," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2013, pp. 167–176.

[14] A. R. Benson, S. Schmit, and R. Schreiber, "Silent error detection in numerical time-stepping schemes," *International Journal of High Performance Computing Applications*, pp. 1–20, 2014.

[15] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "Fti: High performance fault tolerance interface for hybrid systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*. New York, NY, USA: ACM, 2011, pp. 32:1–32:32.

[16] X. Xu, "Large eddy simulation of compressible turbulent pipe flow with heat transfer," *Doctorate Thesis, Iowa State University*, 2003.

[17] J. Shin, M. W. Hall, J. Chame, C. Chen, P. F. Fischer, and P. D. Hovland, "Speeding up nek5000 with autotuning and specialization," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ser. ICS '10. New York, NY, USA: ACM, 2010, pp. 253–262.

[18] S. Habib, V. A. Morozov, H. Finkel, A. Pope, K. Heitmann, K. Kumaran, T. Peterka, J. A. Insley, D. Daniel, P. K. Fasel, N. Frontiere, and Z. Lukic, "The universe at extreme scale: Multi-petaflop sky simulation on the bg/q," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'12)*, 2012, pp. 1–11.

[19] Leonardo A. Bautista-Gomez, "Data Corruption Propagantion on a CFD code," http://leobago.com/projects/sdc/.

[20] L. A. Bautista-Gomez and F. Cappello, "Detecting silent data corruption through data dynamic monitoring for scientific applications." in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'14)*, 2014, pp. 381–382.

[21] S. Di, E. Berrocal, L. Bautista-Gomez, K. Heisey, R. Guptal, and F. Cappello, "Toward effective detection of silent data corruptions for hpc applications," ser. SC '14 - poster, 2014.